CodHoop: A System for Optimizing Big Data Processing

Zakia Asad
The Edward S. Rogers Sr.
Department of Electrical
and Computer Engineering,
University of Toronto, Canada
Email: z.asad@mail.utoronto.ca

Mohammad Asad Rehman Chaudhry*
Soptimizer, Canada
The Edward S. Rogers Sr.
Department of Electrical
and Computer Engineering,
University of Toronto, Canada
Email: mohammad.chaudhry@utoronto.ca

David Malone
Hamilton Institute,
National University of Ireland
Maynooth, Ireland
Email: david.malone@nuim.ie

Abstract—**The rise of the cloud and distributed data-intensive ("Big Data") applications puts pressure on data center networks due to the movement of massive volumes of data. This paper proposes *CodHoop* a system employing network coding techniques, specifically index coding, as a means of dynamically-controlled reduction in volume of communication. Using Hadoop as a representative of this class of applications, a motivating use-case is presented. The proof-of-concept implementation results exhibit an average advantage of 31% compared to vanilla Hadoop implementation which depending on use-case translates to 31% less energy utilization of the equipment, 31% more jobs that run simultaneously, or to a 31% decrease in job completion time.**

I. INTRODUCTION

Recent business insights into datacenter networks evolution forecast an unprecedented growth of data center traffic with 76% of the aggregate traffic not exiting the data center [1]. The increasing migration of applications to the cloud is probably the major driver of this trend: even without a fundamental change in the average communication/computation ratio exhibited by datacenter workloads, increasing the compute density per server by means of virtualization leads to proportional increase in traffic generated per server. Orthogonally though, to the effects of virtualization, the outset of applications crunching and moving large volumes of data - captured by the market-coined term "Big Data applications" - is also foreseen to significantly contribute to higher network utilization/congestion [2].

Complementary to existing approaches for ameliorating the network effects of bringing disruptive (in terms of communication intensity) distributed workloads into the datacenter, we explore in this paper the potential of integrating concepts and techniques of network coding - specifically of index coding - into Hadoop MapReduce. Apache Hadoop [3], an open-source implementation of the MapReduce framework, constitutes a popular and thus good representative of data-intensive workloads. It has been shown to scale-out to thousands of nodes and is used across various domains (from biosciences to commercial analytics). More than half of fourtune 50 companies use Hadoop [4]. Facebook's Hadoop cluster holds more than 100 Petabytes of data which grows at a rate of half Petabytes every day [5]. Focusing on the *shuffle* phase of Hadoop, which is known to be the communication-intensive phase of the application [6], we present in this paper an excerpt of our work on adapting index coding to assess its applicability and efficiency in reducing the volume of communication taking place during the shuffle phase.

It has been shown that mixing techniques such as network coding and index coding can significantly increase the rate of information transfer in a network [7], [8], [9], [10], [8]. We therefore propose a novel index coding based algorithm to improve the rate of information transfer during the Hadoop shuffle phase.

Contributions

We propose CodHoop a system for optimizing big data processing in a data center, and start by presenting a motivating use-case. We have performed a proof-of-concept implementation of CodHoop in a real world data center. We have developed a novel solution for making index coding applicable in wired data center environments, by extending the concept of index coding which only considers wireless environments. We use Hadoop, the most widely used big data processing framework, as our target framework. We have use Grep, an industry standard benchmark, for performance evaluation of CodHoop. The experimental results exhibit an average advantage of 31% compared to the standard Hadoop implementation which depending on use-case translates to 31% less energy utilization of the equipment, 31% more jobs that run simultaneously, or to a 31% decrease in job completion time.

II. MOTIVATING USE-CASE

Before delving into system specification details, we present a use-case with the help of a real, albeit toy, example. The example highlights theft detection in an area by using Hadoop MapReduce framework.

Indoor marijuana growers need lots of electricity for irrigation and power lighting and steal a significant amount of electricity for masking their operations. For example, in
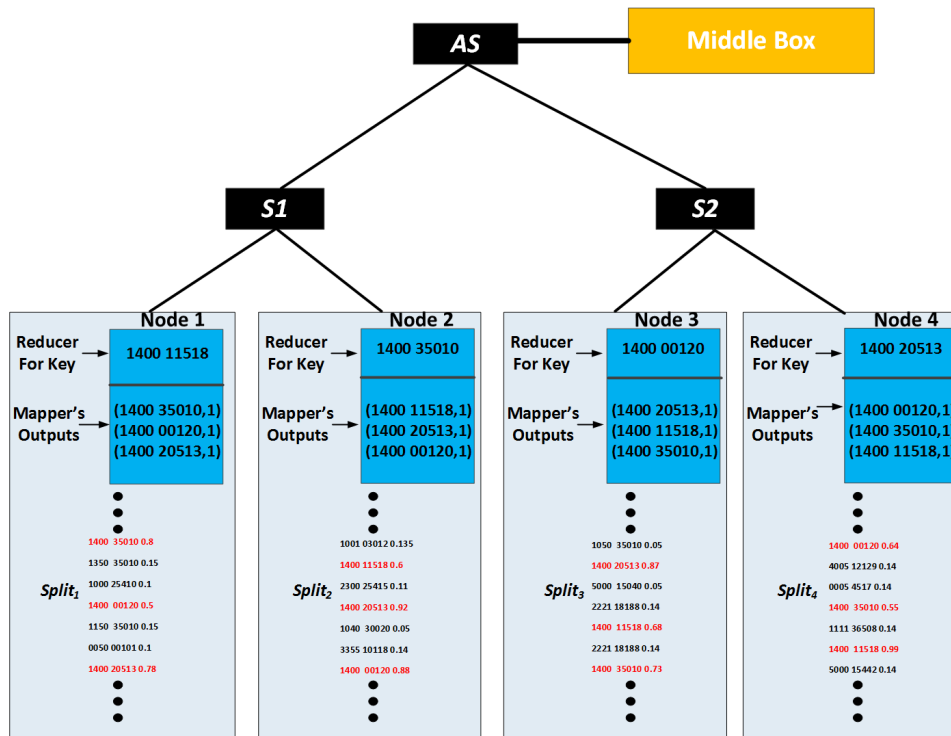
Fig. 1. Topology and mapper/reducer output/input in the Grep application example.

Canada utilities are reported to lose an amount exceeding $500 million from electricity theft mainly due to indoor pot growers. The Canadian province on front end of pot growing is British Columbia with 848 cases that cost the customers staggering $100 million per year [11]. A similar scenario in U.S. is causing CenterPoint Energy in Texas a loss of $450,000 per month. However using illicit activity detection techniques based on analysis and correlation of smart meter data can help in decreasing loss of revenue, for example by detecting unusual patterns of energy usage indicating suspicious activity. An approach to detect illicit electricity usage is to look for unusual high consumption, or unaccounted for electricity consumption in an area. Since the worldwide installation of smart meters is estimated to grow at compound annual growth-rate of 26.6% between 2010 and 2016 [12]. The smart meters usually take frequent readings almost every 15 minutes throughout the day resulting in a amount of data generated to Petabytes per day[13]. This phenomenal growth of data is posing the biggest challenge in sniffing the data for extracting useful insights efficiently. One of the most appropriate options for analyzing such a huge amount of data is distributed processing platform of Hadoop MapReduce.

The example highlights a use-case for detecting unusual high consumption, and unaccounted for electricity consumption in an area by using Hadoop MapReduce framework. The Hadoop job scans the data to count the number of times the power consumption was higher than a threshold. The data used in this example is regenerated and anonymized (for privacy, and confidentiality reasons) from real world smart meter data records accessed via the Irish Social Science Data Archive [14], where the readings were taken every 30 minutes from 5000 smart meters for a period of two years.

Each smart meter reading consists of the following three components meter ID, day and time codes, and power consumed. For example the smart meter reading, 1001 03612 0.135, represents an energy consumption of 0.135, for meter ID 1001, and the reading is taken on 5th of February (036th day of the year) for the 30 minute interval starting at 5:30 am (12th 30 minute interval with first interval starting at 12:00 am).

A MapReduce task usually splits the input into independent chunks which are first processed by the *mappers* placed across different data center nodes in a completely parallel manner. The outputs of the *mappers* are then communicated to the *reducers* which are also placed across different nodes, during the *shuffle* phase, for further processing to complete the job [15]. We note that in a typical real world Hadoop cluster many mappers and reducers share common information. Namely, a data center node(server) often hosts both a mapper and a reducer, whereby the intermediate $< key, value >$ pairs stored by the *mapper* on local storage are accessible to the *reducer* running on the same server without the need for going through the network. This gives rise to ample opportunity to code the packets exchanged during the *shuffle* phase.

A. Hadoop Job

Consider a four nodes Hadoop cluster in a standard data center network architecture as shown in Figure 1. The objec-

tive is to count the total number of instances when the smart meter with ID 1400 reported the power consumption exceeds 0.5 for four different time code and day code considered for this example, i.e., 11518 (April 25th for 30 minute interval starting at 8:30 am), 35010 (December 16th for 30 minute interval starting at 5:30 am), 00120 (January 1st for 30 minute interval starting at 9:30 am), 20513 (July 24th for 30 minute interval starting at 6:00 am).

The input log of smart meter records is divided by Hadoop into four splits ($split_1, \cdots, split_4$), and each split is stored on each of the four *DataNodes* ($node_1, \cdots, node_4$) subscribed to the Hadoop Distributed File System (HDFS), whereby we assume that $split_i$ is stored on $node_i$ as shown in Figure 1.

Each $mapper_i$ residing on $node_i$ parses file $split_i$ and emits the corresponding $< key, value >$ pairs, where a *key* is the smart meter ID along with associated day and time code, and a *value* is the number of times the power consumption exceeds 0.5. In this job the role of the map function is to only extract the parts of the smart meter data log containing the information about the meter ID 1400 for one of the four targeted times and dates.

Before being stored, intermediate $< key, value >$ pairs output by each *mapper* are partitioned such that each *reducer* receives (fetches) pairs corresponding to only one *key*. The reduce function processes the ($< key, value >$) pairs emitted by the *mappers* and counts the total frequency where power consumption exceeds the threshold; for this, four reducer instances are created one at each of the four nodes *DataNodes* ($node_1, \cdots, node_4$). Due to the partitioning strategy, each reducer is responsible for counting the ($< key, value >$) pairs corresponding to a single *key* (e.g. $reducer_2$ residing on $node_2$ is responsible for counting the $< key, value >$ pairs with all keys equal to "1400 35010"). Given the above setting, the output of a mapper's ($< key, value >$ pairs) during the $shuffle$ phase will be delivered to the reducers, e.g., the outputs of the mappers with key *1400 11518* are delivered to the $reducer_1$. Without loss of generality, we assume that a $< key, value >$ pair is communicated by a mapper to a reducer through a single packet transmission, specifically $P_1, \cdots, P_{12}$; where $P_i = (1400 \ 20513, 1) \ for \ i = 3, 5, 7$, $P_j = (1400 \ 00120, 1) \ for \ j = 2, 6, 10$, $P_k = (1400 \ 35010, 1) \ for \ k = 1, 9, 11$ and $P_n = (1400 \ 11518, 1) \ for \ n = 4, 8, 12$.

B. Standard Hadoop mechanism

Using standard Hadoop mechanism, it is easy to find that a total of 10 link-level packet transmissions are required per reducer to fetch all of its desired $< key, value >$ pairs from respective mappers. For example $reducer_1$ residing on $node_1$ is responsible for counting the $< key, value >$ pairs with all keys equal to "1400 11518", this process results in 10 link-level packet transmissions which can be calculated as below:

- Two link-level packet transmissions to fetch packet P_4 from $mapper_2$ residing on $node_2$,
- Four link-level packet transmissions to fetch packet P_8 from $mapper_3$ residing on $node_3$,

- Four link-level packet transmissions to fetch packet P_{12} from $mapper_4$ residing on $node_4$.

It follows then due to symmetry that 40 link-level packet transmissions are required to complete the *shuffle* phase. Note that a total of 16 link-level packet transmissions cross the network bisection ($AS - S_1$, $AS - S_2$) during the *shuffle* phase.

C. Proposed coding based Hadoop mechanism

We note here that a reducer residing on the same node with a mapper has access to this mapper's output without going through the network; for instance, output of $mapper_1$ is locally available to $reducer_1$. We call this locally stored and accessible information as the *side information* available to a reducer, e.g., the *side information* of $reducer_1$ is P_1,P_2 and P_3. Leveraging on this *side information*, we propose coding packets at the $L2 - aggregate$ switch, whereby coding refers to applying a simple $XOR(\oplus)$ function to a set of input packets.

In our example, coding is employed at the bisection of the toy network topology (using middlebox attached to AS switch in Figure 1) and resulting in only following two packets crossing the bisection: $P_2 \oplus P_5 \oplus P_8 \oplus P_{11}$, and $P_3 \oplus P_6 \oplus P_9 \oplus P_{12}$. Each reducer can use its *side information* to decode its required packets from the coded packet it receives. For example, when $reducer_1$ receives $P_2 \oplus P_5 \oplus P_8 \oplus P_{11}$, it can decode its required packet P_8 by $XORing$ the received packet with the packets it already has (P_1,P_2 and P_3), i.e., $P_8 = (P_2 \oplus P_5 \oplus P_8 \oplus P_{11}) \oplus P_1 \oplus P_2 \oplus P_3$. This follows from the fact that although $P_5 = (1400 \ 20513, 1)$ and $P_3 = (1400 \ 20513, 1)$ are generated by different mappers, they contain the same information and hence $P_5 \oplus P_3 = 0$. Similarly, $P_1 \oplus P_{11} = 0$ since $P_1 = (1400 \ 35010, 1)$ and $P_{11} = (1400 \ 35010, 1)$.

Information-Theoretic Equivalence Relation: To explain the decoding process in an intuitive way, we introduce an *information-theoretic equivalence relation*, denoted by $\equiv$. Specifically, two different packets are *information-theoretically equivalent* if they convey the same information, for example although packets P_4,P_8, and P_{12} are different packets but as they carry the same information *(1400 11518,1)*, so we call them information-theoretically equivalent represented by an equivalent A, i.e., $P_4 \equiv P_8 \equiv P_{12} := A$. Similarly, $P_1 \equiv P_9 \equiv P_{11} := B$, $P_2 \equiv P_6 \equiv P_{10} := C$, and $P_3 \equiv P_5 \equiv P_7 := D$. Let's explore how $reducer_1$ would be able to decode its required packet fetched from $mapper_3$. We start by focussing on the coded packet $P_2 \oplus P_5 \oplus P_8 \oplus P_{11}$ received by $reducer_1$:

$$P_2 \oplus P_5 \oplus P_8 \oplus P_{11} \equiv C \oplus D \oplus A \oplus B$$

Then utilizing *side information* available at $reducer_1$:

$$(P_2 \oplus P_5 \oplus P_8 \oplus P_{11}) \oplus P_1 \oplus P_2 \oplus P_3 \equiv C \oplus D \oplus A \oplus B \oplus B \oplus C \oplus D$$

$$= A \equiv P_{11} = (1400 \ 11518, 1)$$

, i.e., the packet required by $reducer_1$ from $mapper_3$.

Together with the packet exchange occurring via the access switches and the transmission of the packets input to the coding function for them to reach the point of coding (aggregation

switch in our example), we find that in this case a total of 36 link-level packet transmissions are required to complete the *shuffle* phase. More specifically:

- Each of the four reducers fetches one record each from the DataNode one-switch away connected via $L2 - switch$ $S_1(S_2)$ incurring 2 link-level packet transmissions per reducer, and a total of 8 link-level packet transmissions for all the reducers,
- 16 link-level packets transmissions, four for each of the four DataNodes to $L2 - Aggregate$ switch(AS),
- A total of 12 link-level packets transmissions to deliver both the coded packets from $L2 - Aggregate$ switch(AS) to all the DataNodes.

Note that by using coding a total of 12 link-level packet transmissions cross the network bisection during the *shuffle* phase, i.e., compared to baseline Hadoop implementation a 25% reduction in network bisection traffic. So our approach compared to current state of the art, depending on the use-case, **translates to 25% less energy utilization of the equipment, 25% more Hadoop jobs that run simultaneously, or to a 25% decrease in job completion time if there is congestion.** The use of identical values for each distinct key generated by a mapper in this example — picked deliberately to ease presentation — favours the efficiency of coding, but obviously may not hold for production Hadoop computations. We generalize the concept of *the coding-based shuffle* beyond this simplifying assumption in the Section IV.

III. RELATED WORK

Many studies have been conducted to optimize the data transfer during the communication-intensive phase [2], [16], [17], spanning from high-level shifting of virtual machines to mitigation of congestion incidents using Hadoop communication reduction mechanisms (combiners) to low-level continuous runtime network optimization in coordination with data-intensive application orchestrators. In comparison to the related work which treat data flow as commodity flow (routing), we utilize the novel concept of coding, and it has been proven that the coding at the network layer can provide the optimal rate of information exchange in many scenarios where the routing can not [18]. Our work is complementary to the redundancy elimination schemes [19] for increasing end-to-end application throughput, and additionally our scheme provides an instantaneous encoding and decoding of flows.

IV. CODHOOP AND ITS COMPONENTS

Our coding scheme for Hadoop MapReduce borrows from novel concepts of the *index coding*, where each client *Wants* some information and has some *side information* [8]. The output of the mapper residing on the same node with a reducer is essentially the reducer's *side information*, whereas the keys that a specific reducer is assigned to process is what it *Wants*.

We introduce three new stages, namely sampler, coder, and preReducer to the traditional Hadoop MapReduce. The primary objective of the sampler is to gather *side information*. Similarly the primary objective of the coder is to code, and

of the preReducer is to decode. The overall architecture is shown in Figure 2, while it shows only two nodes it is in fact replicated across all the nodes.

A. Sampler

We introduce a sampler, residing in the middlebox, which works on the intermediate $< key, value >$ pairs emitted by all the mappers to the partition to gain insight into the *side information* of each physical node. These random records are fetched in the start and in parallel to the *shuffle* phase. This sampling process does not interfere with the *shuffle* process.

B. Coder

The coder is a dedicated software appliance (or middlebox) strategically placed in the datacenter network providing for wire-speed packet payload processing, and re-injection of coded packets into the network. The coder determines which packets to code, and how to code. An instance of the *extended index coding* problem is defined by a coding server (middlebox), a set $X = \{c_1, \ldots, c_m\}$ of m clients(nodes), and a set $P = \{p_1, \ldots, p_n\}$ of n packets that need to be delivered to the clients. Each client c_i is interested in a certain subset of packets known as its *Want* set $W_i \subseteq P$, and has access to some *side information* known as its *Has* set $H_i \subseteq P$. Note, a client might not possess any *side information*, i.e., its *Has* set might be empty. The server can transmit the packets in P, or their combinations (coded packets) to the clients via a point to point wired network. For efficiency, the only coding operations that the server is required to perform are restricted to $\oplus$ i.e., operations over $GF(2)$. The goal is to find a transmission scheme that requires the minimum number of link-level packet transmissions to satisfy the requests of all clients.

The *coder* performs the following three functions:

1) **Format Decision Making:** Based on initial sampled data from each physical machine, this step decides on coding format. *Coder* treats a $< key, value >$ pair as a *data chunk*. Note that for a specific Hadoop job some bytes of $< key, value >$ might contain more mutual information than others. Since coding exploits the mutual information between different file splits residing on different physical machines; the decision on the coding format is based on comparing the coding advantage, reduction in traffic volume across the network, for coding on different chunks of the data packet.

2) **Coding:** This step performs the extended index coding and extends techniques due to Chaudhry et. al. [10] on anticipated bytes conforming to the format decided in previous step. To enable our middlebox keep processing at the line-rate, the algorithm ensures that the coded packets are *instantaneously decodable* at the receiver nodes i.e., just based on the packets received and without any need of buffering a set of incoming packets.

3) **Packaging:** This step packs the outcome of the coding process into a custom packet payload format and then re-inject into the network towards the reducer nodes.
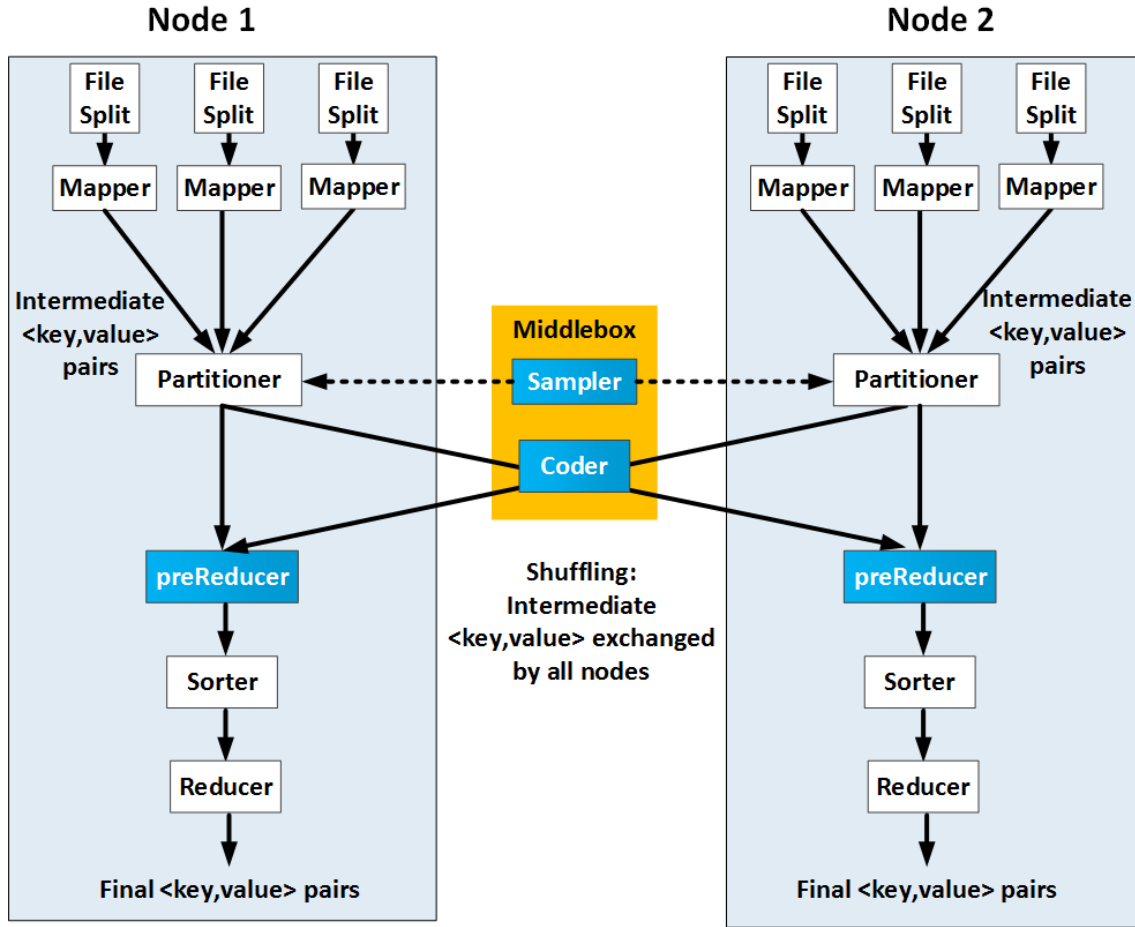
Fig. 2. CodHoop Architecture.

C. PreReducer

The role of the *preReducer* component is to ingest custom packets sent by the *Coder*, decode their payloads data and extract the $<key, value>$ pairs which are to be input to the standard Hadoop *Sorter*.

V. EXPERIMENTAL RESULTS

We have prototyped parts of the system in a data center as an initial proof of concept implementation. Our testbed consisted of 8 identical servers (96 processor cores). Each server was equipped with twelve x86_64 cores, 128 GB of RAM, and a single 1 TB hard disk drive. The servers were connected in a typical datacenter configuration. The components were implemented using Java [20], and Octave [21]. All the servers were running Red Hat Enterprise Linux 6.2 operating system.

We present here preliminary results obtained by running a *Grep* (Global Regular Expression) benchmark. *Grep* represents generic pattern matching and searching on a data set. Our data set consisted of on an organization's data-logs, whereby the goal was to calculate the frequency of occurrence of different types of events in the data-log input.

We use the following metrics for quantitative evaluation:

- *Job Gain*, defined as the increase (in %) in the number of parallel Hadoop jobs that can be run simultaneously with *coding based shuffle* compared to the number of parallel Hadoop jobs achieved by standard Hadoop.
- *Utilization Ratio*, defined as the ratio of link-level packet transmissions when employing *coding based shuffle* to the number of link-level packet transmission incurred by the standard Hadoop implementation.

Our experimental study shows that for both of the tested benchmarks, the overhead to implement *the coding-based shuffle* (in terms of transmission of extra bookkeeping data in packet headers) was less than 4%. Table I shows the results across the two metrics for the two benchmarks, where gain was found to be over 25% in both cases.

Noting the fact that our coding-based scheme just requires $XORing$ of packets which is computationally very fast fast operation and given larger memory bandwidth of the servers, we were able to process closer to line rate. Specifically in the experimental setup, even during the worst case scenario, the throughput of the coder was 809 $Mbps$ on a 1 $Gbps$ link.

Hadoop Job	Job Gain	Utilization Ratio
Grep	31%	0.69

TABLE I

Job Gain AND *Utilization Ratio* USING PROPOSED CODING BASED SHUFFLE.

VI. CONCLUSION

In this paper, we have introduced for the first time to our knowledge a network middlebox service that employs *extended index coding* for reducing the volume of communication in Hadoop. The results reported from initial prototyping are promising compared to the current state of the art.

VII. ACKNOWLEDGEMENTS

REFERENCES

[1] http://www.cisco.com/.
[2] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. *SIGCOMM-Computer Communication Review*, 41(4):98, 2011.
[3] http://hadoop.apache.org/.
[4] http://www.prnewswire.com/news-releases/altiors-altrastar—hadoop-storage-accelerator-and-optimizer-now-certified-on-cdh4-clouderas-distribution-including-apache-hadoop-version-4-183906141.html.
[5] https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920/.
[6] A. Curtis, K. Wonho, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *IEEE INFOCOM 2011*.
[7] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network Information Flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.
[8] M.A.R. Chaudhry, Z. Asad, A. Sprintson, and M. Langberg. Finding sparse solutions for the index coding problem. In *IEEE GLOBECOM 2011*.
[9] M.A.R. Chaudhry and A. Sprintson. Efficient algorithms for index coding. In *IEEE INFOCOM Workshops 2008*.
[10] M.A.R. Chaudhry, Z. Asad, A. Sprintson, and M. Langberg. On the Complementary Index Coding Problem. In *IEEE ISIT 2011*.
[11] Utility ami analytics for the smart grid 2013-2020: Applications, markets and strategies. Technical report, GTM Research, June 2013.
[12] http://www.intelligentutility.com/magazine/article/253959/6027-million-installed-smart-meters-globally-2016.
[13] http://www.accenture.com/us-en/blogs/technology-blog/archive/2014/06/19/hadoop-based-distributed-parallel-computing-improving-the-performance-of-ami-data-analytics.aspx.
[14] www.ucd.ie/issda/.
[15] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.
[16] P. Costa, A. Donnelly, A. Rowstron, and G. OShea. Camdoop: Exploiting in-network aggregation for big data applications. In *USENIX NSDI*, volume 12, 2012.
[17] X. Lin, Z. Meng, C. Xu, and M. Wang. A practical performance model for hadoop mapreduce. In *2012 IEEE International Conference on Cluster Computing Workshops*.
[18] R. Ahlswede, N. Cai, S. Li, and R. Yeung. Network information flow. *IEEE Transactions on Information Theory,*, 46(4):1204–1216, 2000.
[19] Ashok Anand, Vyas Sekar, and Aditya Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 87–98. ACM, 2009.
[20] http://www.java.com/.
[21] John W. Eaton, David Bateman, and Soren Hauberg. *GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform, 2009. ISBN 1441413006.